Role and Attribute Based Collaborative Administration of Intra-Tenant Cloud IaaS

(Invited Paper)

Xin Jin
Department of Computer Science
Institute for Cyber Security
University of Texas at San Antonio
San Antonio, Texas

Ram Krishnan
Department of Electrical and Computer Engineering
Institute for Cyber Security
University of Texas at San Antonio
San Antonio, Texas

Ravi Sandhu
Department of Computer Science
Institute for Cyber Security
University of Texas at San Antonio
San Antonio, Texas

***Abstract*—Cloud Infrastructure as a Service (IaaS), where traditional IT infrastructure resources such as compute, storage and networking are owned by a cloud service provider (CSP) and offered as on-demand virtual resources to customers (tenants), is the fastest maturing service model in cloud computing. The transformation of physical resources into virtual offers great flexibility to CSP customers including network based remote collaborative administration. This flexibility can be fully availed only if complemented by commensurately flexible access control to the customers remote IT resources by the customer's IT users. Since customer policies in this regard can vary greatly, the CSP needs a flexible model to accommodate diverse policy requirements. In this paper, we investigate attribute-based access control (ABAC) in cloud IaaS. In ABAC, access requests are evaluated based on the attributes of cloud tenant users and those of objects such as virtual machines, storage volumes, networks, etc. We investigate the access control models supported by commercial IaaS providers such as Amazon AWS and opensource OpenStack, as well as other models in the literature, which mostly use role-based access control (RBAC). We demonstrate their limitations and motivate the need for ABAC support to realize the true potential of IaaS. Building on prior published ABAC models we define a formal ABAC model suitable for IaaS. As proof-of-concept we implement this model in OpenStack, a widely-used open source cloud IaaS software platform. We discuss enforcement alternatives in this context and partially evaluate their performance.**

***Index Terms*—attribute based access control, cloud computing, infrastructure as a service**

I. INTRODUCTION

Cloud computing is revolutionizing the way businesses and governments manage their information technology (IT) assets. Infrastructure as a Service (IaaS) cloud, where traditional IT infrastructure such as compute resources complemented by storage and networking capabilities are owned and operated by a cloud service provider (CSP) and offered as an on-demand service to its customers (tenants), is being rapidly adopted by many organizations [1], [5]. Many newer companies such as Netflix, Dropbox and Instagram have eschewed development of proprietary IT infrastructure in favor of CSPs. Established companies such as SAP, GE, Adobe and Domino's Pizza, are increasingly migrating to the cloud. In this paper, we use the following terminology. We have CSP, organizations and tenants. An *organization* becomes a *tenant* of a particular CSP when it signs up for services with that CSP. For our purpose, it suffices to assume that an organization has at most one tenant at a CSP. We use the terms tenant and *customer* interchangeably.

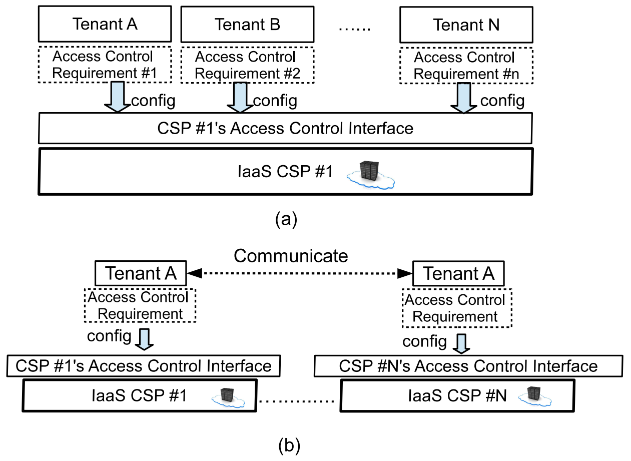

Tenant A
Tenant B
......
Tenant N
Access Control Requirement #1
Access Control Requirement #2
Access Control Requirement #n
config
CSP #1's Access Control Interface
IaaS CSP #1
(a)
Communicate
Tenant A
Tenant A
Access Control Requirement
config
CSP #1's Access Control Interface
CSP #N's Access Control Interface
IaaS CSP #1
IaaS CSP #N
(b)

Fig. 1: *Access Control in IaaS Cloud*

Although the functional aspects of IaaS are maturing, the security issues involving this technology are not yet fully understood. Security is often cited as a leading concern in moving to cloud [10], [26], [27], for reasons including uncertainty in continued control over a customer's assets and lack of interoperability between the customer and CSP, and across different CSPs. In particular, when an organization uses the cloud, it faces unfamiliar and non-standard abstractions of access control facilities provided by the CSP over its virtualized resources (compute, storage, networking, etc.). Several challenges arise when an IT infrastructure is outsourced to the cloud. We illustrate them through figures 1(a) and 1(b). Figure 1(a) shows that CSP#1 has multiple tenants. Figure 1(b) shows that tenant A is a customer of CSP#1 to CSP#N. What access control requirements arise in this scenario? Consider the resources in IaaS including virtual machines (VM), storage and network. In a traditional enterprise data center, an organization specifies policies for its IT personnel over its assets including who can access server rooms, maintain servers, add or remove server capacity, start, stop or take a snapshot of the server, establish a

COLLABORATECOM 2014, October 22-25, Miami, United States
Copyright © 2014 ICST
DOI 10.4108/icst.collaboratecom.2014.257591

network, modify network configurations, add storage, backup, connect a storage volume to a server, etc. When moving to IaaS cloud, these resources become virtual and remote. Access control policies in the physical world are in part achieved via physical keys, access cards and fingerprints, and will need to be comparably specified and enforced in the cloud. That is, in IaaS, there is a need to mimic policies enforcing both physical and digital controls in traditional IT operations.

Two major issues emerge in context of figure 1(a). Before moving to cloud, each organization has its own in-house access control policies. However, when it becomes a tenant of CSP #1, the organization must re-think their native policies in terms of the CSP's access control facilities. A dual problem manifests for the CSP. Each customer will want to configure their own access control policies which may be vastly different from that of others. With an unknown number of potential customers, it is unrealistic to pre-design and implement all kinds of access control models in the cloud or design one by one on demand [27]. The cloud platform should provide a flexible and intuitive access control framework such that customers can easily configure their own access control policies. Furthermore, in order to distribute their resources (e.g., for availability), some organizations may be tenants of multiple CSPs as in figure 1(b). This poses an additional problem of dealing with multiple different access control interfaces and integrating them.

Current access control models for IaaS in the academic literature and industry are mostly built on role-based access control (RBAC), sometimes extended with attributes. These models fall short with respect to the above challenges, since RBAC caters more for ease of management as opposed to flexibility and fine-grained control.

Consider the following scenarios. Bob, an IT person in tenant A creates a VM. As the creator, he has complete rights over this VM. However, he may wish to grant selected rights over this VM to other IT users in tenant A. Consider Alice, an IT person in tenant B who creates a set of VMs that need to be highly available. First, she wants to ensure that these VMs are managed only by users with "networkOperator" role. Next, she wants to ensure that not all VMs are in "stopped," "underMaintenance" or "underMigration" state simultaneously to guarantee availability. Alice also wants to create a storage volume to store sensitive information, so she wants to ensure that this storage volume can only be attached to VMs with an image with the right patches and security updates. That is, a "sensitive" volume can only be attached to "hardened" VMs.

These scenarios illustrate the diverse access control needs that may arise in IaaS. The original RBAC models have been consistently extended in various directions with novel features to meet various demands [12]. The large body of RBAC literature over the last two decades has identified compelling new features that are necessary. We believe that a unified model, which is capable to cover existing RBAC-related models as well as new models which enhance RBAC, is required for cloud IaaS to respond to the demand of various enterprise infrastructure applications. To our best knowledge, such a model has not yet been studied.

Attribute-based access control (ABAC) is a natural and intuitive candidate for this purpose, and is gaining traction in enterprises [2]. In ABAC, access control decisions are based on attributes of various entities such as users, subjects and objects. Sufficient abstractions can be built on top of ABAC in order to closely mimic the access control abstractions expected by each tenant. A sufficiently flexible ABAC engine at the CSP-side can be configured to enforce each tenant's access control expectations. Thus the requirements of figure 1(a) can be met. Figure 1(b) requires standardization of the ABAC IaaS capabilities supported by different CSPs, which may emerge over time [14].

Although there has been considerable work in ABAC [3], [14], [17], [18], [21], [30], [32], what is lacking today is an ABAC framework for IaaS that is intuitive and easy to administer and use, yet with formal foundations to provide ease of extensions to cover RBAC-related models and adding new features to RBAC. This is vital for successful adoption of ABAC in IaaS given the complex real-world nature of the domain. Defining such an ABAC model is an interesting and challenging research task. We address this problem by systematically evaluating major access control models in IaaS cloud in academia and industry and showing limitations of these models compared with the core requirements of access control in cloud IaaS. We then present a formal framework of ABAC which satisfies those requirements. To demonstrate practicality, we conduct implementation and partial evaluation of the models in the prominent IaaS platform OpenStack [4]. OpenStack is a robust open-source IaaS software for building public, private or hybrid clouds that is developed and maintained by a vibrant community with participation from more than 200 world-leading organizations.

The rest of this paper is organized as follows. Section II motivates the design of IaaS access models. Section III discusses why existing ABAC models are not sufficient. Section IV defines our proposed ABAC model for IaaS cloud. Section V introduces enforcement models for ABAC in OpenStack. Section VI presents experimental results and section VII gives our conclusions.

II. Motivation

In this section, we first discuss some basic terminology and a typical process for initiating a tenant account in the cloud. We then present two examples of tenants of a cloud IaaS provider. Based on these we summarize the core requirements for access control in IaaS. We then review access control models in two leading cloud IaaS platforms: OpenStack (Grizzly release) and Amazon Web Services (AWS). For our purpose, the concept of an "account" in AWS and a "project" in OpenStack are the same as "tenant." In our discussion below, we uniformly use the term tenant. We also review models discussed in the IaaS access control literature.

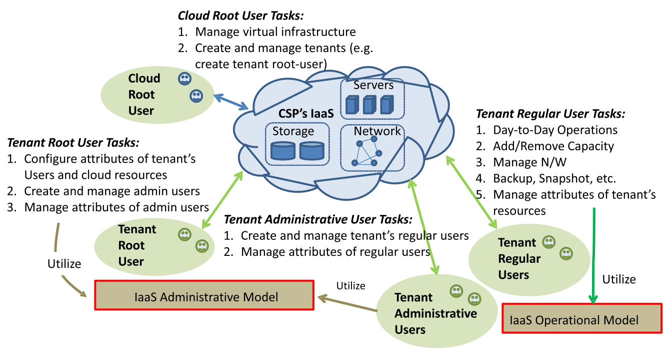

Cloud Root User Tasks:
1. Manage virtual infrastructure
2. Create and manage tenants (e.g. create tenant root-user)
Cloud Root User
CSP's IaaS
Servers
Storage
Network
Tenant Root User Tasks:
1. Configure attributes of tenant's Users and cloud resources
2. Create and manage admin users
3. Manage attributes of admin users
Tenant Regular User Tasks:
1. Day-to-Day Operations
2. Add/Remove Capacity
3. Manage N/W
4. Backup, Snapshot, etc.
5. Manage attributes of tenant's resources
Tenant Administrative User Tasks:
1. Create and manage tenant's regular users
2. Manage attributes of regular users
Tenant Root User
Tenant Regular Users
Utilize
Utilize
Utilize
IaaS Administrative Model
Tenant Administrative Users
IaaS Operational Model

Fig. 2: *Access Control Challenges In IaaS Cloud*

A. Access Control Approach for Cloud IaaS

The users who interact with cloud IaaS in a management or administrative capacity are categorized into different types (shown in four ovals in figure 2). A **cloud root user** is a user who manages cloud resources for the CSP. For ease of presentation we assume there is a single all-powerful cloud root user who is described as carrying out various functions manually. In practice many functions of the cloud root user would be automated and triggered by commitment of payment, in keeping with the self-service on-demand paradigm.

On the tenant side, we have there three types of IT users. By IT user we mean a user in an organization that provides IT support to that organization. A **tenant root user** represents an IT user who has root access to the tenant. For ease of presentation, we assume that for each tenant, there is only one root user who has full permissions in the tenant. The tenant root user is created by the cloud root user. A **tenant admin user** represents an administrative IT user with administrative permissions in the tenant. Administrative permissions allow management of regular IT users (discussed below) and their attributes in a tenant. A **tenant regular user** is a regular IT user with permissions for standard IT operations such as creating and deleting virtual machines, storage volumes, networks, etc., on the tenant's behalf. Note that in figure 2 an administrative model is necessary to guide the tasks of tenant root and administrative IT users while an operational model is necessary for managing the tasks of regular IT users. The administrative model facilitates creating and updating attributes while the operational model facilitates specifying authorization policies that control the actions of regular IT users. We emphasize that non-IT users of a tenant who only interact with the cloud for using the VMs and other services are not considered in figure 2. They do not manage any cloud IaaS resources and are controlled by access control mechanisms within the VMs and within applications running in VMs.

A simplified general process for an organization to move to cloud is as follows. In order to use cloud services, the first step is that an organization's representative (say Alice) obtains an account from the CSP typically via some automated process which is a surrogate for the cloud root user. Thereby the organization becomes a tenant of the CSP with Alice as that tenant's root user. Now it is not practical for Alice to create and manage all the resources herself. Instead, in the second step Alice sets up tenant specific access control and administrative policies using the CSP-provided facilities, and creates some number of tenant admin users. Then, the tenant admin users create regular IT users and administer their attributes. Finally, regular IT users can then create and manage virtual resources as per the policies specified by the tenant root user and attribute values administered by tenant admin users.

B. Example Scenarios

Let us consider two tenants of a CSP: **TechU** and **iGame**.

TechU. A university called TechU wants to create a data center in the cloud. Bob leads the project and becomes the root user of this tenant. Bob then specifies and configures the administrative and operational policies.The university contains certain number of colleges under which there are several departments. The university, each college and each department maintain certain amount of resources for different purposes. For consistency, all departments, colleges and the university are called domains. Each domain contains different types of resources. There are instances, volumes, networks and images. Each domain requires certain number of different types of resources for different services such as Web, Email, LibApp and SMSApp. IT architects are added as tenant regular users so that they can manage cloud resources. They are assigned with one or multiple service and domain pairs. The following administrative and operational policies need to be specified:

- TP1. Bob has all the permissions (i.e., create instances, add users, assign roles, etc.) within the tenant. ITManager is a role which can add and delete tenant regular users and assign their role to be ITArchitect. ITManager can further assign ITArchitect to any domain and service pairs.
- TP2. When new resources are created, the creating user can only assign those resources to the domain and service that she is assigned to.
- TP3. ITArchitects can access a resources only if they are assigned to the same domain and service as the resource.
- TP4. A user can choose to activate any service and domain pair(s) for each login session with the server.

iGame. In this game development company, there are users (i.e., tenant regular user) with different roles such as ServerIT, StorageIT and Manager. Some of these ServerIT users are assigned to the project called DeepLearning. All virtual machines (VM) that serve as game servers are assigned to one of the types from Tablet, TV, Phone and Laptop. The VMs are configured with different CPUs, memories and disks depending on their type. The purpose is to deliver smooth performance to different client devices as there could be more phone users than TV or Laptop users. In addition to the type, game servers are also assigned to different countries because the number of users from different countries vary and the content can also vary with the locality and local law. Additional VMs are created for the purpose of running machine learning algorithms to study the habit of game payers.

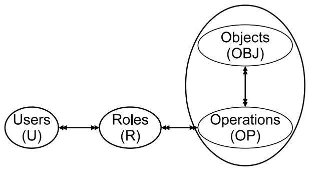
Those virtual machines are also assigned with type (e.g., user habit can be different on Laptop and Phone) and country. User logs are saved in storage media which are assigned to different countries and time ranges (we consider logical time such as Morning, Noon, Afternoon and Evening) because those factors can impact user habits. The following operational policies need to be specified:

- GP1. ServerIT users can only start and stop game servers which are assigned to the same country as the user and the user must be assigned to the game which is running on the server. In order for ServerIT users to start and stop servers for learning user habits, they must be assigned to a project called DeepLearning. StorageIT users can resize the storage of the country to which the users are assigned. Manager inherits permissions from ServerIT. In addition, Managers can take a snapshot of game servers for games they manage.
- GP2. When a ServerIT user creates a new server, it should be labeled with the same country as that of the creator.
- GP3. ServerIT users can be assigned to multiple countries. However, a user can only have access to one country for each login session with the server.

C. Core Requirements

Based on our illustration in section II-A and the example scenarios in section II-B, we summarize a list of core requirements that are specific for access control in cloud IaaS.

- Req 1. *Tenants' full control over their access control design and specification.* Each tenant should have full control over their access control policies and management of their users, which include tenant-specific administrative and authorization capabilities. This implies two things. First, each tenant root user is able to specify its own attribute design and create access control abstractions (e.g. an RBAC abstraction built over the CSP-provided ABAC engine) that are suitable for that tenant. Second, all operations (i.e., create a virtual machine, add a user, etc.) within a tenant can be controlled by that tenant instead of depending on the cloud root user.
- Req 2. *Simple yet flexible administrative policy.* As there is a tenant root user who configures and manages administrative aspects, the administrative model should provide certain ease in its configuration. At the same time, this model should be able to specify fine-grained control over tenant regular users and their information (i.e., sensitive information such as role or attributes that are used for authorization).
- Req 3. *Flexible operational model with potential to cover RBAC-based models.* Our analysis of existing cloud IaaS providers reveals that most access control abstractions are RBAC-based. Note we say *RBAC-based* since roles are typically augmented with additional parameters for flexibility. Thus the third requirement is that the operational model supported by the cloud provider should provide sufficient flexibility to specify policies based on different aspects including system defined attributes such as time and location, and tenant defined attributes such as roles, department, groups, sensitivity, etc. This allows each tenant to design their own abstractions of access control that is intuitive to them.
- Req 4. *Strong formal foundations.* Formal foundations allow rigorous security specification and analysis enabling precise understanding of the capabilities of the access control system of IaaS.



(a) Access Control for OpenStack

- U, O, OP, R, EXP and PER represent the finite set of existing users, objects, operations, roles, expressions and permissions respectively.
- $PER \subseteq OP \times O$, represents an operation-object assignment.
- $URT \subseteq U \times R$, represents a user-role assignment.
- $OPE \subseteq OP \times EXP$, represents an operation-expression assignment.

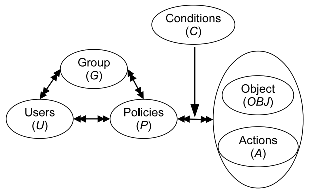
(b) Formal Model

Fig. 3: *OpenStack Access Control*

D. Existing Access Control in Cloud IaaS

1) OpenStack: OpenStack is an open source IaaS software adopted by many cloud service and technology providers such as Rackspace, IBM, Dell and RedHat. The structure of OpenStack access control model is shown in figure 3(a). The major components in each tenant include *Users*, *Objects*, *Roles*, *Operations*, *Permissions*, and *Expressions*. A permission is an operation on an object. Each user may be assigned to multiple roles, such as "professor" or "manager". Each operation is associated with a boolean expression specified using the usual $\wedge$ and $\vee$ operators on terms of the form r and $\bar{\text{r}}$ where r is a role. The expression is evaluated for a user by interpreting r to be true if the user is assigned with role r and $\bar{\text{r}}$ to be true if user is not assigned with role r. E.g., consider "compute : create_instance: $\mathrm{r}_1 \wedge \mathrm{r}_2 \wedge \bar{\mathrm{r}}_3$". It says that the user is authorized to perform the compute : create_instance operation if he is assigned with roles r_1 and r_2 and is not assigned with role r_3. If a user tries to operate on an object, the policy check is as follows: the user's roles in the same tenant as the object should satisfy the expression associated with the operation. E.g., a user is assigned with roles $\{\mathrm{r}_1, \mathrm{r}_2\}$ in tenant t_1 and roles $\{\mathrm{r}_1, \mathrm{r}_3\}$ in tenant t_2. According to the above policy, he is authorized to perform compute : create_instance operation in t_1 but not in t_2. The formal model is summarized in figure 3(b).

This model is RBAC-based and thus has some formal foundations (Req 4). However, it does not satisfy Req 1, Req 2 and Req 3. More specifically, OpenStack access control has two major issues. Firstly, tenants are not provided with full control over access control policies. All tenants share the same policy for all OpenStack components such as Nova (compute), Keystone (identity and access management) and Glance (VM image repository), and these policies can only be configured by CSP root users. There is no mechanism to customize access control policies for individual tenants. Clearly,



(a) Access Control Model for AWS

- U, O, C, G, A and P represent the finite set of existing users, objects, conditions, groups, actions and policies.
- $UPA \subseteq U \times P$, represents a many to many user-policy assignment.
- $UGA \subseteq U \times G$, represents a many to many user-group assignment.
- $PGA \subseteq P \times G$, represents a many to many policy-group assignment.

(b) Formal Model

Fig. 4: *Amazon Web Service Access Control*

OpenStack cannot intuitively configure the policies in tenants iGame and TechU. Secondly, access control in OpenStack is coarse grained because only operation level authorization is supported. If an operation such as compute : stop_instance is authorized to a user, that user can stop any VM instance in the tenant. Authorization of this operation for particular tenant VMs cannot be specified. A recent release of OpenStack introduces the concept of domain, which is roughly equal to the concept of tenant with regard to administration, and supports domain to define their own roles. The operational model does not change. Thus it does not satisfy Req 3. Although OpenStack has evolved (it has a release cycle of 6 months), changes to the access control model [29] do not address the above limitations so far.

2) Amazon Web Service: AWS [22], [31] is the commercially dominant cloud IaaS platform. Example services include elastic compute cloud (EC2), simple storage service (S3) and elastic block storage (EBS). We discuss the AWS Identity and Access Management (IAM) component which concerns access control as related to the above cloud services.

The access control model structure is shown in figure 4(a). The major components in each tenant are *Users*, *Objects*, *Groups*, *Policies*, *Actions*, *Conditions*, *User-Group-Assignment*, *User-Policy-Assignment* and *Policy-Group-Assignment*. A *permission* is defined in the format of *Action* on *Object* under certain *Conditions*. *Conditions* are in the form of key-value pairs. Each key-value pair can be one of following types including String, Numeric, Date & Time, Boolean, and IP-address. For example, the condition "DateLessThan: {aws : CurrentTime : "2013-09-01T00:00:00Z"}" uses the Date & Time type DateLessThan condition restricting the requests to be made before Sep 1, 2013 [31]. A group is a similar concept as role in RBAC. It is associated with a set of policies which define a set of permissions. Users can either be assigned to groups or directly to policies. Each policy is specified using the policy specification language provided by AWS. Each policy consists of a number of statements which contain a description of the requests they apply to, plus an effect, which may be allow or deny. Each statement contains lists of actions, lists of resources and lists of principals plus a number of conditions which must be met. Principals can be users, groups or roles. As roles are used mainly for cross account access, they are out of scope for this paper. The formal model is summarized in figure 4(b).

In summary, AWS is also RBAC-driven with fixed attributes such as time and location in addition to the group (role) attribute. Since those attributes are unalterable, it does not satisfy Req 3. Although AWS supports adding tags (i.e., key and value pairs) to resources (e.g., EC2, S3), tags cannot be added to users. In addition, tags are only used for managing resources and cannot be used in access control policies. Furthermore, commercial models (including OpenStack) are developed in an ad hoc manner. They are designed as simple as possible at the beginning and extended based on incremental customer requirements. The nature of this development process limits the formal foundation of the model and thus the system may need cumbersome extensions in the future or even force a redesign to fulfill new requirements. Hence Req 4 is not satisfied.

Evidently, AWS cannot configure TP3 in TechU and GP1 in iGame unless tags can be used in authorization policy. In addition, AWS cannot intuitively configure TP1, TP2 and TP4 in TechU and GP2 and GP3 in iGame because the tagging operation is not constrained so the user can tag the resource with any key and value pairs.

3) Other RBAC-Driven IaaS Models: Most of the access control models for IaaS in the literature are RBAC-driven. The classical RBAC model is not sufficient. Firstly, RBAC cannot configure TP1 in TechU because its administrative models such as [23] are inadequate. Secondly, RBAC can only configure TP3 and TP4 for tenant TechU and GP1 and GP3 for iGame, but the configurations are cumbersome because of two major problems. The first limitation is role explosion. In order to configure these policies, a large set of roles needs to be defined. The second problem is role-permission assignment. The virtual resources may be dynamically created and deleted. In traditional RBAC, role-permission assignments have to be reconfigured manually for created or deleted objects.

Wu *et al.* [31] designed and implemented access control as a service (ACaaS) based on RBAC to extend the AWS access control model. ACaaS introduced role hierarchies, sessions, constraints and an administration model. Domain based access controls (*dCloud*) [24], [25] were proposed based on the original RBAC model. The general idea is to group related resources and users into a domain so that administration can be delegated to each domain. Daniel *et al.* provided an authorization system to control the execution of virtual machines (VMs) to ensure that only administrators and owners could access them [20]. Berger *et al.* [7] proposed an authorization model based on both RBAC and security labels to control access to shared data, VMs, and network resources. Almutairi *et al.* [6] proposed a distributed access control architecture for cloud computing, focussing on enforcement aspects where the actual policy is RBAC-based. Chadwick *et al.* [9] proposed a federated identity management system for cloud. Takabi *et al.* [28] proposed a comprehensive security framework for cloud computing environments illustrating a big picture of security requirements in cloud. Although some customized RBAC can configure the policies for iGame and TechU, a common drawback of RBAC-related models above is that they do not satisfy Req 1 and/or Req 3.

III. Existing ABAC Models

In order to be flexible, the ABAC models for cloud IaaS require many of the well-known constructs in access control. For example, to configure GP3 in iGame tenant, it is necessary to distinguish user and subject (processes, connections and activities in systems that perform operations on behalf of users) and specify policy to constrain subject attributes based on user attributes. However, this user-subject separation is not widely supported in most ABAC models. We further review related work as follows.

Firstly, there are limited number of initial works on using ABAC for cloud in general as opposed to specifically for IaaS. Cha *et al.* [8] proposed ABAC in cloud computing environment. Iqbal *et al.* [15] proposed semantic-enhanced ABAC for cloud services. Danwei *et al.* [11] proposed access control for cloud service based on UCON. However, these works are neither focused on IaaS access control nor present a formal ABAC model. In addition, they did not present the functionalities for the entire lifecycle of tenant management (e.g., tenant creation, policy configuration, administration, etc.). Secondly, general ABAC has been studied in many areas. Formal ABAC models are proposed [19], [21], [30]. However, [21] does not incorporate subject and user distinction. [19] defines credentials to specify user role assignment delegation rules and it is focussed on an attribute called role. [30] uses set theory for authorization policy specification which is only one part of ABAC. Many policy languages have also been proposed. While important, a policy language by itself is not sufficient for ABAC. For example, XACML [3] provides formats for request and policy specification. Its main focus is how to process authorization using policy structure including policy integration and conflict resolution. Key-Policy Attribute-Based Encryption (KP-ABE) [13] associates policy trees instead of lists of attributes with private keys. [33] presented a new ABE scheme called Attribute-Based Encryption with Attribute Lattice (ABE-AL) that provides an efficient approach to implement comparison operations between attribute values on a poset derived from attribute lattice. In general, ABE is a method for securely enforcing ABAC policies on data sharing and access control. Due to the complexity and cost, the access control policies (AND, OR, NOT, etc.) included in these techniques are very limited in expressive power. Thus, XACML and ABE can only express policy TP3 in TechU and GP1 in iGame. They cannot express other policies as they do not distinguish user and subject and are based on the assumption that user and object attributes are already provided. ABAC enforcement models are also discussed. For example, [32] proposed to use ABAC in web service. In summary, we find that an ABAC framework equipped with necessary features to satisfy the core requirements of access control in IaaS cloud is currently lacking.

IV. Formal Models

In this section, we provide formal specifications of two related models: the operational model $\mathrm{IaaS_{op}}$ and the administrative model $\mathrm{IaaS_{ad}}$. $\mathrm{IaaS_{op}}$ enables specification of authorization policy for day to day operations of the tenant regular IT users. $\mathrm{IaaS_{ad}}$ enables specification of administrative policy for user management.

We first design the $\mathrm{IaaS_{op}}$ model based on the recently proposed unified ABAC_α [17] model with necessary modifications. The general idea of $\mathrm{IaaS_{op}}$ is as follows. Users, subjects and objects are associated with attributes. Subject and object attributes are set and modified by users and subjects respectively. The modification of subject and object attributes are guided by constraint policies specified in the model. A subject is able to perform an action on an object (e.g. stop a VM) as allowed by the authorization policies in the model. For the purpose of this paper, the ABAC_α model serves as an adequate $\mathrm{IaaS_{op}}$ model as long as one necessary change is made: objects are partitioned into types and each type of object is associated with a set of object attributes. In the original ABAC_α model, all objects are associated with the same set of attributes. We thus adopt ABAC_α for $\mathrm{IaaS_{op}}$ and integrate the necessary change. With this modification, Req 3 and Req 4 (section II-C) are satisfied.

We then design the $\mathrm{IaaS_{ad}}$ model also based on the recently proposed user-attribute assignment model GURA [16]. In this model, administrative permissions such as add and assign user attributes are associated with administrative roles. Tenant administrative IT users are then assigned these roles.While this model is simple, being role-based, it provides fine-grained access control over user attributes. However, GURA doe not address user management and relationship between different types of users. Hence, we adopt GURA as our base administrative model and extend it with the necessary features (i.e., user provisioning and deletion). With this new $\mathrm{IaaS_{ad}}$ model Req 2 is satisfied.

Note that after combining the two models, Req 1 is also satisfied. In this paper we only deal with access control issues where there is a strict separation between tenants. Each tenant maintains their specific $\mathrm{IaaS_{op}}$ and $\mathrm{IaaS_{ad}}$ policy and only users within the same tenant are authorized to access resources in that tenant. For example, tenant administrative users can only create tenant regular users for the tenant she belongs to. When defining our model, we do not explicitly show the mappings between model components (i.e., users, attributes) and the tenants to simplify presentation. Even though we focus on isolated tenants in this paper, cross-tenant access control can be achieved by suitably extending our models.

A. The Operational Model $IaaS_{op}$

1) Components: The structure of $\mathrm{IaaS_{op}}$ is shown in the right part of figure 5. The $\mathrm{IaaS_{op}}$ model is configured by the tenant root user. We use "configure" to signify the operation of system architects who design the elements in the system based on formal models. The major components are regular users (TReU), subjects (S), objects (O), user attributes (UA), subject attributes (SA), object attributes (OA), operations (PER), subject ($\mathrm{ConstrSub}$) and object ($\mathrm{ConstrObj}$) attribute constraint and authorization policies (Authz).

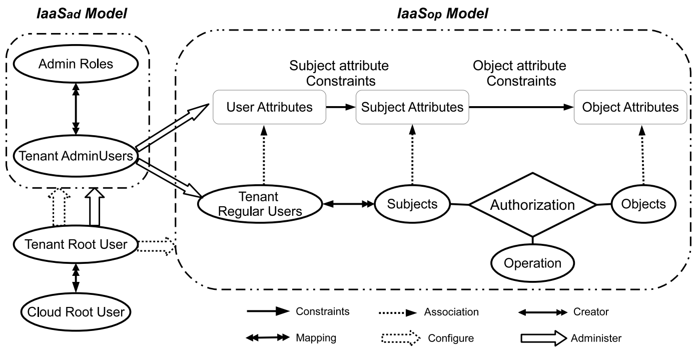


Fig. 5: $\mathrm{IaaS_{op}}$ *and* $\mathrm{IaaS_{ad}}$ *Access Control*

An *attribute* is a function which takes an entity and returns certain properties of the entity. Each attribute is associated with a finite set of atomic values as its scope. There are two types of attributes: set valued and atomic valued. The major difference is set valued attributes can take multiple values from their scope while atomic valued attributes take a single value from their scope. Example set valued attributes are `role` and `division` and example atomic valued attributes are `clearance` and `level`.

A *user* is an entity which interacts with the cloud. We have introduced cloud root user, tenant root user, tenant administrative user and tenant regular user in section II-A. User attributes reflect the properties of users. In this paper, only regular users are associated with attributes since we employ ABAC only for the operational component of IaaS. A *subject* is a program or process created by a user to access resources on behalf of that user. Only the creator can terminate a subject. For example, when a user creates a connection from his mobile phone to the cloud, the connection is a subject. He can also create another concurrent subject from his laptop. A subject carries attributes which can be used for authentication and authorization. Examples are `ip`, `timestamp` and `networktype` (public, private, etc.). Besides those, there may be attributes inherited from user attributes. In some systems, subjects are associated with a signed credential of the users' information and can be encoded in a token (e.g., token in OpenStack [4] or temporal token issued by Security Token Services (STS) in AWS [1]). The cloud authenticates and authorizes all requests submitted by this subject based on information included in the token or access key and secret key pair. Subjects created by a user may take attributes and values that differ from that of its user. *Subject attribute constraint policy* specifies the constraints on subject attributes when users create subjects and set values for subject attributes. For example, a user may be assigned to the following groups {web, email, app} in a tenant. The user get different permissions for each group. Each time the user logs in, he can choose to activate permissions within certain groups instead of all the groups. In this way, user accesses the tenant with least privilege. The specification language is defined in [17].

Objects represent the virtual resources in cloud. Examples are virtual machines, virtual networks, images, volumes and storages. Objects are created by subjects on behalf of users. Objects are also associated with attributes and those attributes are set and modified by their owner who creates them. Different types of objects may be associated with different sets of object attributes. For example, volumes may be associated with `size` and `attachedVM` attributes while it is not meaningful to associate virtual machines with these attributes. When a user sets or modifies the attributes of objects, there are also constraints. *Object attribute constraint policy* specifies the constraints on the values that object attributes may take. An example object attribute constraints policy may require that when a subject creates a volume, the volume should be labelled with the same `division` (or `department`) as the subject and the volume's owner is set to the subject's creator.

An *operation* represents an access mode on objects. Operations are defined by the CSP and will vary across different CSPs. For example, operations on virtual machines include create, start, stop and resize. Operations on images include upload and list.

Authorization policy specifies policies for evaluating requests made by subjects (on behalf of regular IT users). It is specified based on attribute values of the involved subject and object. It returns either **true** or **false** meaning the request is authorized or rejected. For example, if a user requests to stop a virtual machine, the user and the virtual machine should be of the same `division`.

2) Formal Definition: The formal operational ($\mathrm{IaaS_{op}}$) model is summarized in table I. This model is configured

TABLE I: *Basic Sets and Functions for* $\mathrm{IaaS_{op}}$ *Model*

TReU, S and O represent finite sets of *existing* regular users, subjects and objects respectively.

UA, SA and OA represent finite sets of user, subject and object attribute functions respectively.

$\mathrm{objType}$: $\mathrm{O} \rightarrow \mathrm{OT}$. For each object, $\mathrm{objType}$ gives its type.

$\forall\ t \in \mathrm{OT}$, O_t = {obj | obj $\in$ O $\wedge$ t = objType(obj)}, represents objects of type t.

oaType: $\mathrm{OA}\rightarrow 2^{\mathrm{OT}}$. For each object attribute, oaType gives its types.

$\forall\ t \in \mathrm{OT}$, OA_t = {oa | oa $\in$ OA $\wedge$ $t \in$ oaType(oa)}, represents object attributes of type t.

$\mathrm{SubCreator}$: $S \rightarrow U$. For each subject $\mathrm{SubCreator}$ gives its creator.

For each att in $\mathrm{UA} \cup \mathrm{SA} \cup \mathrm{OA}$, SCOPE_{att} represents the attribute's scope, a finite set of *atomic* values.

$\mathrm{attType}$: $\mathrm{UA} \cup \mathrm{SA} \cup \mathrm{OA} \rightarrow$ {set, atomic}. It specifies attributes as set or atomic valued.

PER represents finite set of operations.

Each attribute function maps elements in TReU, S and O to atomic or set values.

$$\forall ua \in \mathrm{UA}.\, ua : \mathrm{TReU} \rightarrow \begin{cases} \mathrm{SCOPE}_{ua} & \text{if } \mathrm{attType}(ua) = \text{atomic} \\ 2^{\mathrm{SCOPE}_{ua}} & \text{if } \mathrm{attType}(ua) = \text{set} \end{cases}$$

$$\forall sa \in \mathrm{SA}.\, sa : \mathrm{S} \rightarrow \begin{cases} \mathrm{SCOPE}_{sa} & \text{if } \mathrm{attType}(sa) = \text{atomic} \\ 2^{\mathrm{SCOPE}_{sa}} & \text{if } \mathrm{attType}(sa) = \text{set} \end{cases}$$

$$\forall t \in \mathrm{OT}. \forall oa \in \mathrm{OA}_t . oa : \mathrm{O}_t \rightarrow \begin{cases} \mathrm{SCOPE}_{oa} & \text{if } \mathrm{attType}(oa) = \text{atomic} \\ 2^{\mathrm{SCOPE}_{oa}} & \text{if } \mathrm{attType}(oa) = \text{set} \end{cases}$$

TABLE II: *Selected Operations For Tenant Regular Users*

Operations	**Updates**
1.1 createSubject(req:TReU, sub:NAME, saset:SASET)	$S' = S \cup$ sub, for each (sa, val) $\in$ saset, sa(sub) = val, SubCreator(s) = req
1.2 createObject(sub:S, obj:NAME, oaset:OASET_t, t: OT)	$O' = O \cup \{$obj$\}$, objType(obj) = t, for each (oa, val) $\in$ oaset, oa(obj) = val
1.3 Operations(sub:S, obj:O)	None

by tenant root users. The *basic sets and functions* in $\mathrm{IaaS_{op}}$ model are as follows: TReU, S and O represent finite sets of tenant regular users, subjects and objects respectively. There is one distinguished attribute for object, objType, which maps objects to their respective types. OT represents the scope of this function and thus O_t represents the set of objects of type t. We define a finite set of object types based on the current architecture of cloud IaaS. For example, OT = {vm, file, image, network, volume}. UA, SA and OA represent finite sets of user, subject and object attributes respectively. oaType is a function mapping each object attribute to the types of objects it applies. For each t in OT, OA_t represents the attributes defined for objects of type t. These could be atomic or set valued as determined by the type of the attribute function (attType). For each attribute, SCOPE represents the finite set of atomic values it can take. SubCreator is a distinguished attribute which maps each subject to the user who creates it. Finally, PER represents finite set of operations.

There are three *policy configuration points* in the $\mathrm{IaaS_{op}}$ model. Authz, SubConstr and ObjConstr represent finite sets of authorization, subject and object attribute constraints policies. We need a language to express these policies which we adopt from [17]. The subject attribute constraint policy is specified by comparing the proposed value of subject attributes with the attribute values of the creating user. The object attribute constraint policy is given as a comparison between the proposed object attribute value and subject attribute value. Authorization policy is specified by comparison between attributes of the involved subject and object. In all cases these comparisons can be combined by logical conjunction, disjunction and negation operators. Examples are provided in the next section.

3) Operations for tenant regular users: Operations can be submitted by tenant regular users to the cloud. The cloud system updates state according to the operation if it is authorized. The precondition of each operation is defined as in [17]. We call the user who submits the operation the "requester". The following operations are authorized to regular users if the evaluation result from authorization policy is **true**. We briefly introduce the format of each operation in table II. Operation 1.1 creates a subject, operation 1.2 creates an object and operation 1.3 represents any of the regular operations on resources such as starting a virtual machine, creating a volume, etc. In these operations, NAME is an abstract data type denoting identifiers of various entities. SASET represents the data type in which each element represents an attribute assignment for all subject attributes. Similarly, OASET_t represent the data type in which each element represents an attribute assignment for all object attributes for object type t. In next section, we will give examples on how to evaluate these operations and also discuss how to configure and administer the $\mathrm{IaaS_{op}}$ model.

B. The Administrative Model IaaS$_{ad}$

1) Components: Recall that cloud root user (CRU) and tenant root user (TRU) have been defined in section II-A. The structure of $\mathrm{IaaS_{ad}}$ model is shown in the left part of figure 5. This model is configured by tenant root user and administered by tenant root users. Here "administer" signifies operations such as creating users and modifying user role assignment. The major components are tenant administrative users (TAU), administrative roles (AR) and user role assignment (UAR). *Administrative roles* are associated with administrative permissions such as add, delete and assign user attributes. *Administrative users* are associated with administrative roles and thus obtain the associated permissions.

2) Formal Definition: The formal administrative model $\mathrm{IaaS_{ad}}$ is summarized in part I in table III. The *basic sets and functions* are CRU, TRU, TAU, AR and UAR. CRU and TRU represent the cloud root user and tenant root user respectively. TAU represents the set of administrative users, AR represents a finite set of administrative roles, and UAR represents user administrative role assignment.

There is one *configuration point* for $\mathrm{IaaS_{ad}}$ model which is *administrative policies*. They specify the condition under which certain administrative roles can modify user attributes. The precondition is specified based on the attribute value of the user whose attributes are to be modified. AdminPolicy represents finite set of administrative policies. Again, we need a language for specifying these policies which we adopt from [16]. For each attribute att in UA, $\mathtt{can\_add}_{\mathtt{att}}$ is a set containing tuples in the format of (ar, condition, values) where ar is one of the administrative roles, condition is a boolean expression specified using the current values of attributes of the regular IT users, and values represents a set of value that can be added. It means that administrative role ar can add (more operations will be introduced in section IV-B3) any value from values to the attribute att of user whose attributes satisfy the precondition condition. can_add is defined for set-valued attributes. Similarly, can_delete is defined for set-valued attributes representing policies for delete permission. Finally, can_assign is defined for atomic-valued attributes. The above policy comes from GURA model. In $\mathrm{IaaS_{ad}}$, another policy is needed to control the adding and deletion of tenant regular users. Certain administrative roles are allowed to add and delete tenant regular users. Thus, a can_adduser and a can_deleteuser relations are defined in AdminPolicy to control operations of adding and deleting tenant regular users,

TABLE III: *Formal Definition For* $\mathrm{IaaS_{ad}}$ *Model*

Part I. Basic Sets and Functions	
CRU, TRU represent the cloud root user and tenant root user respectively.	
TAU represents finite set of tenant administrative users.	
AR represents a set of administrative roles and UAR represent user-role assignment, i.e., UAR $\subseteq$ TAU$\times$ AR.	
Part II. Operations	
Operations	Updates
1. Operations for Cloud Root User	
1.1 createTenant(req:CRU, tenant:NAME)	T′ = T ∪ {tenant}
1.2 createRootUser(req:CRU, u:NAME, tenant:T)	TRU = ∅, TRU = {u}
2. Operations for Tenant Root User	
2.1 createUserAttr(req:TRU, ua:NAME, type: {set, atomic})	UA′ = UA ∪ {ua}, attType(ua) = type
2.2 createSubAttr(req:TRU, sa:NAME, type: {set, atomic})	SA′ = SA ∪ {sa}, attType(sa) = type
2.3 addSubConstr (req:TRU, policy:POLICY)	SubConstr′ = SubConstr ∪ {policy}
2.4 createObjAttr (req:TRU, oa:NAME, type: {set, atomic}, oat:OT)	OA′ = OA ∪ {oa}, attType(oa) = type, oaType(oa) = oat
2.5 addObjConstr (req:TRU, policy:POLICY)	ObjConstr′ = ObjConstr ∪ {policy}
2.6 addAuthz (req:TRU, policy:POLICY)	Authz′ = Authz ∪ {policy}
2.7 createAdminRole(req:TRU, adminrole:NAME)	AR′ = AR ∪ {adminrole}
2.8 createAdminPolicy(req:TRU, policy:POLICY)	AdminPolicy′ = AdminPolicy ∪ {policy}
2.9 addAminUserRole(req:TRU, u:TReU, r:AR)	UAR′ = UAR ∪ {(u, r)}
3. Operations for Tenant Administrative Users [16]	
3.1 addUser(req:TAU, u:NAME)	TReU′ = TReU ∪ {u}
3.2 add(req:TAU, u:TReU, att:UA, value:SCOPE$_{att}$)	att(u)′ = att(u) ∪ {value}
3.3 delete(req:TAU, u:TReU, att:UA, value:SCOPE$_{att}$)	att(u)′ = att(u) \ {value}
3.4 assign(req:TAU, u:TReU, att:UA, value:SCOPE$_{att}$)	att(u)′ = value

where can_adduser $\subseteq$ AR and can_deleteuser $\subseteq$ AR. They are both subsets of AR representing the roles which can perform the corresponding operation.

3) Operations: We define a set of operations to maintain the sets and relations defined above and in $\mathrm{IaaS_{op}}$ model. We provide a selected list of the operations in part II in table III (we provide a complete list in appendix A). We briefly introduce them here.

Category I. Operations For Cloud Root User. Firstly, we define operations for cloud root user. These operations will only be authorized if the requester (req) is the cloud root user. That is, req = CRU, where req is the formal parameter and represents the actual requester in each operation. For simplicity, we assume that a tenant can only have one tenant root user. Operation 1.1 creates a new tenant in the system and operation 1.2 assigns a tenant root user.

Category II. Operations For Tenant Root User. Secondly, we define operations for tenant root user. They are authorized if and only if the requester is the tenant root user, i.e., req $\in$ TRU. Operation 2.1 adds a set-valued or atomic-valued user attribute. Operation 2.2 adds a set-valued or atomic-valued subject attribute. Operation 2.3 creates a subject attribute constraints policy. POLICY is an abstract data type whose elements represent identifiers of policies (authorization policy, subject attribute constraints policy, etc.) that may appear in $\mathrm{IaaS_{op}}$ system. Operation 2.4 adds a set-valued or atomic-valued object attribute. Operation 2.5 adds an object attribute constraints policy at object creation and modification time. Operation 2.6 creates the authorization policy for regular users. Operation 2.7 creates an administrative role and operation 2.8 creates policies for administrative roles (we adopt the structure and specification language from [16]). Operation 2.9 adds a user-role assignment.

Category III. Operations For Tenant Administrative Users. The following operations are allowed by tenant root user or administrative users if they are assigned with appropriate administrative roles. We briefly introduce the format and evaluation of each operation. Operation 3.1 adds a regular user. The precondition of this operation (and also operation for deleting a tenant regular user) is authorized by can_adduser and can_deleteuser relations. The precondition for addUser(req, u) request is (similar precondition can be defined for deleteUser operation):

$$(\exists(req, r) \in UAR.r \in \text{can\_adduser}) \vee req = \mathrm{TRU}$$

For example, if can_adduser = {manager, director}, then any tenant administrative users with manager or director role or tenant root user can add tenant regular users to the tenant. Operation 3.2 add(req, target_user, `att`, value), where req is the requester, target_user is the user whose attribute is to be added to, `att` represents the attribute to be modified, and value represents the value to be added. Similarly, operation 3.3 delete(req, target_user, `att`, value) and operation 3.4 assign(req, target_user, `att`, value) are defined. The preconditions for these operations are defined in corresponding administrative policies. We provide a case study of these operations (including operations introduced for $\mathrm{IaaS_{op}}$) in appendix A.

V. Proof Of Concept

We demonstrate practicality of the models of the previous section by a proof-of-concept OpenStack implementation. We briefly introduce the authorization and authentication components in OpenStack and propose three different enforcement models. OpenStack contains the following components: Nova, Swift, Glance, Cinder, Keystone, Quantum and Horizon. Each component acts as a service which communicate with each other via message queues and hence are loosely-coupled. Nova provides virtual servers upon demand. Swift provides object storage. Glance provides a catalog and repository for virtual disk images. Horizon provides a modular web-based user interface for all OpenStack services. Quantum provides network connectivity as a service between interface devices managed by other OpenStack services. Cinder provides persistent block storage to VMs. Keystone provides authentication and authorization for all the OpenStack services. In our discussion, we focus on Keystone and Nova.

A. *Access Control in OpenStack*

Authorization in OpenStack is enforced by a Policy Enforcement Point in each component. Keystone is the component that stores user information including tenant and role assignments. Keystone provides the user information in the format of a token signed by Keystone's private key. All other components obtain the public key of Keystone when added as a service. Thus, the public key of Keystone is only distributed to trusted components. They verify the user information by decoding the user's token. Other components then authorize the user based on the user information provided by the token. Generally, Keystone is the policy information point (PIP) where user information is stored and each component has its own policy enforcement point (PEP), policy decision point (PDP), policy administration point (PAP), and a PIP where respective object attributes are stored.

A general authorization process for Nova component is illustrated in figure 6. A user sends the user name and password to Keystone for authentication and obtains service end point addresses for various OpenStack services. Keystone then verifies the provided user name and password and generates a token with signed user data. Keystone sends the token back to the user together with the service endpoints (e.g., address for Nova service). The user then sends a request to the Nova service using the token and request details (e.g., operation, arguments). The Nova service's PEP component verifies and validates that the provided token is not revoked by communicating with Keystone. The PEP component then retrieves object data from local PIP and decodes the token with Keystone's public key. User and object data together with the request are sent to the PDP component. The PDP retrieves policy from local files and evaluate the request. A result of `true` or `false` is returned meaning that the request is either authorized or denied.

B. *Enforcement Models*

We consider three different enforcement models. The structure of the first enforcement model is shown in figure 7. This method maintains the original architecture of OpenStack. Keystone stores user attributes definitions, user attribute assignments, subject attributes definitions, subject attribute assignment and subject attribute constraints policy. When a user authenticates through Keystone and tries to create a subject with suggested values for each subject attribute, Keystone verifies the suggested attributes against subject attributes constraints policy and the creating user attributes. Then

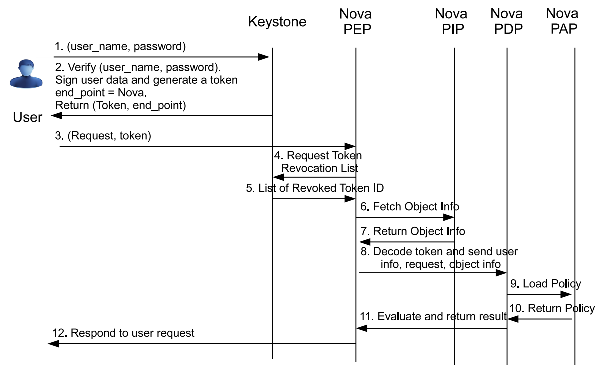

Keystone
Nova PEP
Nova PIP
Nova PDP
Nova PAP
User
1. (user_name, password)
2. Verify (user_name, password). Sign user data and generate a token end_point = Nova. Return (Token, end_point)
3. (Request, token)
4. Request Token Revocation List
5. List of Revoked Token ID
6. Fetch Object Info
7. Return Object Info
8. Decode token and send user info, request, object info
9. Load Policy
10. Return Policy
11. Evaluate and return result
12. Respond to user request

Fig. 6: *OpenStack Authorization Using Asymmetric Keys*

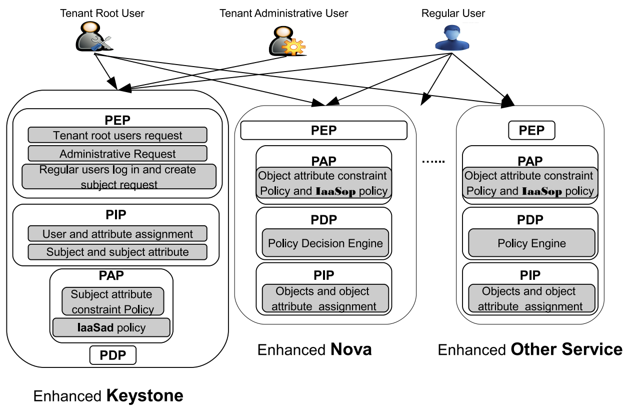

Tenant Root User
Tenant Administrative User
Regular User
PEP
Tenant root users request
Administrative Request
Regular users log in and create subject request
PIP
User and attribute assignment
Subject and subject attribute
PAP
Subject attribute constraint Policy
IaaSad policy
PDP
Enhanced Keystone
PEP
PAP
Object attribute constraint Policy and IaaSop policy
PDP
Policy Decision Engine
PIP
Objects and object attribute assignment
Enhanced Nova
......
PEP
PAP
Object attribute constraint Policy and IaaSop policy
PDP
Policy Engine
PIP
Objects and object attribute assignment
Enhanced Other Service

Fig. 7: *Proposed ABAC Enforcement Model I*

Keystone generates a token by signing the suggested subject attributes. The administration policy is stored, enforced and decided in Keystone. Components excluding Keystone store object attributes, object attribute assignments and policies for authorization and object attribute constraints policy.

Enforcement Model II defines a centralized policy engine. The structure is different from that of enforcement model I only in the part shown in figure 8(a). We design a separate component called PolicyEngine. It is the central point for policy storage and authorization evaluation. All other components, instead of calling local policy evaluation engine, forward their authorization request (containing details about the request and user token) to this component. Included items in the forwarded request are: subject attributes, object attributes and operation. With the centralized design, all policies for all tenants are stored centrally in a single component. Thereby policy administration is decoupled from the policy enforcement. Object attribute constraints is expressed using authorization policy. However, this enforcement model sacrifices performance for convenience. There is a network latency because each request is sent to the PolicyEngine as a REST call.

We propose a third enforcement model III shown in figure 8(b). It is different from Enforcement model II only in that a centralized object attribute store is provided, where all object attributes are stored. When each component enforces their

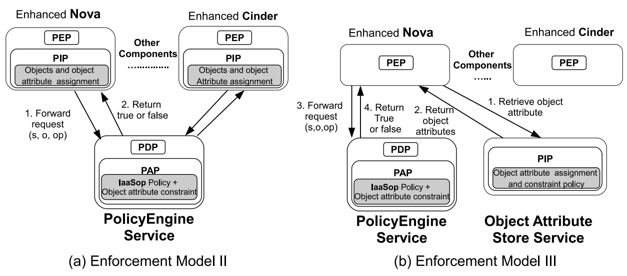

Enhanced Nova
PEP
PIP
Objects and object attribute assignment
Other Components
Enhanced Cinder
PEP
PIP
Objects and object Attribute assignment
1. Forward request (s, o, op)
2. Return true or false
PDP
PAP
IaaSop Policy + Object attribute constraint
PolicyEngine Service
(a) Enforcement Model II
Enhanced Nova
PEP
Other Components
Enhanced Cinder
PEP
3. Forward request (s,o,op)
4. Return True or false
2. Return object attributes
1. Retrieve object attribute
PDP
PAP
IaaSop Policy + Object attribute constraint
PIP
Object attribute assignment and constraint policy
PolicyEngine Service
Object Attribute Store Service
(b) Enforcement Model III

Fig. 8: *Proposed ABAC Enforcement Model II and III*

policies, there are two ways to interact with object attribute store: (1) Each component retrieves object attributes from the object attribute store and forwards the request to the PolicyEngine. (2) The PolicyEngine receives request from other components and retrieves the object attributes from the centralized object attribute store.

VI. PERFORMANCE EVALUATION

A. Experiment Content

We have completed a first stage implementation of ABAC for the Nova and Keystone service components of OpenStack. We understand that the efficiency of enforcing an access control model depends on many factors, such as PolicyEngine, number of requests, number of attributes and so on. However, we found that the time increase for token generation in Keystone and network latency introduced by the centralized PolicyEngine in enforcement model II are crucial in evaluating the overall performance of enforcing ABAC. Thus, the experiments in this paper fall in two parts.

Part I. Time increase for token generation in Keystone with or without additional user attributes (If there are no user attribute, it represents the original RBAC implementation in OpenStack). As user attributes are included in the token, it requires longer time for token generation (remember that a token is a signed user credential). For simplicity, we ignore subject attribute constraint policy in this experiment. We test the response time of token generation for cases with 0, 5, 10, 15 and 20 user attributes where "0" means the RBAC implementation in OpenStack. User attributes in the database are stored as $(\texttt{attname}, value, tenant)$ tuples. We send concurrent requests to keystone using Keystone client command and measure the average response time on client side.

Part II. We evaluate the network latency introduced by the centralized PolicyEngine in enforcement model II. The latency is introduced by forwarding the package which contains user token, object attributes and operation. Thus, we measure the average time for the Nova server to send the request to PolicyEngine and receive a result. We change the number of user attributes and concurrent requests.

B. Experiment Environment and Results

Our experiments are based on a private cloud shown in figure 9. It is installed on four physical machines. We install

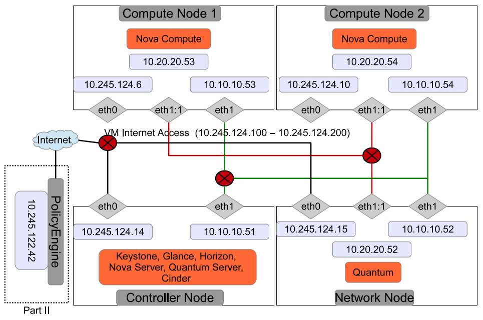


Fig. 9: *OpenStack Installation On Physical Machines*

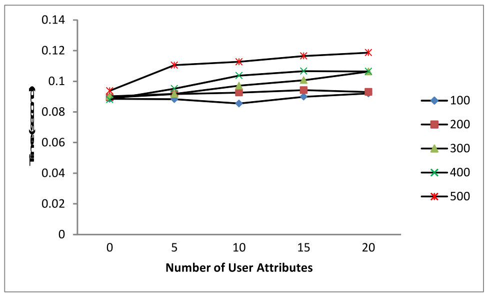


Fig. 10: *Average time for token generation in Keystone*

two compute nodes, one networking node and one controller node. The configuration of controller node and network node is: 24 cores CPU, 24 GB RAM and 1 TB Disk and the configuration of the two Nova compute nodes is: 16 cores CPU, 98 GB RAM and 1TB Disk. There are three networks in this installation: (1) the green line on network interface eth1 shows the administration network which connects different components of OpenStack; (2) the red lines on network interface $\mathrm{eth1:1}$ shows the data network which connects virtual machines with the Internet and (3) the black line on the eth0 network interface shows the access to the Internet which is only accessible by Controller node and Networking node. In experiment part II, we install the centralized PolicyEngine on another machine which has dual core CPU, 4GB RAM and 10 GB disk.

The result for **Part I** is shown in figure 10. A first observation is that given the same concurrent request, the average time for token generation increases with the number of user attributes. This is caused by the increase in the length of user data to be signed by Keystone. As each token contains all user attributes, the signing process and transmission takes longer to finish. However, we can see that the increase is not significant. The time is increased by 20% when the number of user attribute increases from 0 to 20.

The result for **Part II** is shown in figure 11. It can be seen that the networking latency increases with the number of user attributes as data to be forwarded to the PolicyEngine component becomes larger. The latency increases with the number of concurrent request even with the same number of user attributes. This is due to the reason that the PolicyEngine is installed on a machine with limited computing power than

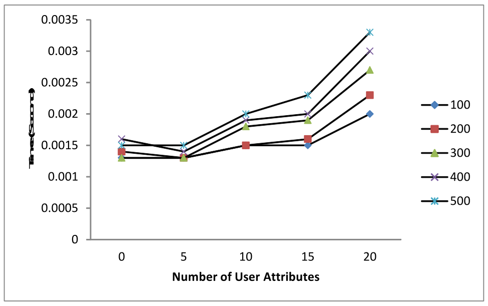


Fig. 11: *Average time for Nova Communicating with PolicyEngine*

the machines we installed OpenStack. The waiting time for getting a policy decision becomes larger when there are too many requests to be evaluated. The average time for request with 20 attributes in 500 concurrent requests is almost twice the time of that with no attributes. We can also observe that the time increase becomes faster with the number of concurrent request when the user attribute increases. However, our implementation of PolicyEngine is not highly optimized. Furthermore, 20 attributes for access control decisions is a big stretch in practice.

VII. Conclusion and Future Work

In this paper, we proposed an ABAC framework for access control in cloud IaaS. We studied existing models from industry and academic literature and motivated the need for ABAC by showing practical examples and limitations of the existing models. ABAC is suitable for cloud IaaS because of its flexibility and potential in unifying RBAC-related and role-driven models. We provided formal models for the operational and administrative aspects of our ABAC framework for cloud IaaS. Based on the proposed ABAC framework, we designed enforcement models based on the open source cloud platform OpenStack. We then partially evaluated the performance of our proposed enforcement models. In the future, we plan to work on policy analysis of ABAC and administration models. In addition, we plan to study the structure of the policy and improve the throughput of the PolicyEngine component.

Acknowledgment

This work is partially supported by the NSF (CNS-1111925) and AFOSR MURI grants (FA9550-08-1-0265).

References

[1] Amazon web services. http://aws.amazon.com.

[2] Attributes are now "how we role". http://www.avatier.com/products/identity-management/resources/gartner-iam-2020-predictions/.

[3] OASIS, Extensible access control markup language (XACML), v2.0.

[4] Openstack. http://www.openstack.org/.

[5] Rackspace customers. http://stories.rackspace.com/customers. 2013.

[6] A. Almutairi, M. Sarfraz, S. Basalamah, W. Aref, and A. Ghafoor. A distributed access control architecture for cloud computing. *IEEE Software*, 2012.

[7] S. et al Berger. Security for the cloud infrastructure: Trusted virtual data center implementation.

[8] J. Cha, B.and Seo and J. Kim. Design of attribute-based access control in cloud computing environment. In *Int. Conf. on IT Conv. and Sec.*, pages 41–50. Springer, 2012.

[9] D. W. Chadwick, M. Casenove, and K. Siu. My private cloud–granting federated access to cloud resources. *Journal of Cloud Computing*, 2013.

[10] S. Crago, K. Dunn, and P. et al Eads. Heterogeneous cloud computing. In *2011 IEEE CLUSTER*, pages 378–385.

[11] Chen Danwei, Huang Xiuli, and Ren Xunyi. Access control of cloud service based on UCON. In *Cloud Computing*. Springer, 2009.

[12] L. Fuchs, G. Pernul, and R. Sandhu. Roles in information security: A survey and classification of the research area. *Comp. and Secur.*, 2011.

[13] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *ACM CCS*, pages 89–98, 2006.

[14] V. C. Hu and D. Ferraiolo et al. Guide to Attribute Based Access Control (ABAC) Definition and Considerations. In *NIST SP 800-162*, 2013.

[15] Z. Iqbal and J. Noll. Towards semantic-enhanced attribute-based access control for cloud services. In *IEEE TrustCom*, 2012.

[16] X. Jin, R. Krishnan, and R. Sandhu. A role-based administration model for attributes. In *ACM WSRAS*, 2012.

[17] X. Jin, R. Krishnan, and R. Sandhu. A unified attribute-based access control model covering DAC, MAC and RBAC. In *DBSEC*. 2012.

[18] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *IEEE Symp S&P*, 2002.

[19] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust management framework. In *2002 IEEE S&P*.

[20] Daniel Nurmi and Richard et al Wolski. The eucalyptus open-source cloud-computing system. In *CCGRID*, pages 124–131. IEEE, 2009.

[21] J. Park and R. Sandhu. The UCON$_{ABC}$ usage control model. *ACM TISSEC*, pages 128–174, 2004.

[22] David Power, Mark Slaymaker, and Andrew Simpson. On the modelling and analysis of amazon web services access policies. Technical Report RR-09-15, Oxford University Computing Laboratory, November 2009.

[23] R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based administration of roles. *ACM TISSEC*, pages 105–135, 1999.

[24] D. Shin, H. Akkan, W. Claycomb, and K. Kim. Toward role-based provisioning and access control for infrastructure as a service (IaaS). *J. Internet Services and App*, 2011.

[25] Dongwan Shin and Hakan Akkan. Domain-based virtualized resource management in cloud computing. In *IEEE CollaborateCom*, 2010.

[26] S Subashini and V Kavitha. A survey on security issues in service delivery models of cloud computing. *J. of Net. and Com. App.*, 2011.

[27] Hassan T., James BD J., and Gail-Joon A. Security and privacy challenges in cloud computing environments. *IEEE S & P*, 2010.

[28] H. Takabi, J. BD Joshi, and G. J. Ahn. Securecloud: Towards a comprehensive security framework for cloud computing environments. In *IEEE COMPSACW*, 2010.

[29] Bo Tang and Ravi Sandhu. Extending openstack access control with domain trust. In *2014 NSS*.

[30] L. Wang, D. Wijesekera, and S. Jajodia. A logic-based framework for attribute based access control. In *ACM workshop FMSE*, 2004.

[31] Ruoyu Wu, Xinwen Zhang, Gail-Joon Ahn, Hadi Sharifi, and Haiyong Xie. Design and implementation of access control as a service for iaas cloud. *SCIENCE*, 2(3):pp–115, 2013.

[32] E. Yuan and J. Tong. Attributed based access control (ABAC) for web services. In *IEEE ICWS*, 2005.

[33] Yan Zhu, Di Ma, Chang-Jun Hu, and Dijiang Huang. How to use attribute-based encryption to implement role-based access control in the cloud. In *Int Workshop on Security in Cloud Computing*, 2013.

Appendix

We show the sequence of operations to configure policies in tenant TechU.

$$\mathsf{createTenant}(\text{Alice}, TechEdu)$$
$$\mathsf{createRootUser}(\text{Alice}, \text{Bob}, TechEdu)$$
$$\mathsf{createUserAttr}(\text{Bob}, \texttt{org\_service}, set)$$
$$\mathsf{createSubAttr}(\text{Bob}, \texttt{sorg\_service}, set)$$
$$\mathsf{addSubConstr}(\text{Bob}, policy)$$

where $policy$ is:

$$\text{ConstrSub}(u, s, \{(\texttt{sorg\_service}, val)\}) \equiv val \subset \texttt{org\_service}(u)$$

$$\begin{aligned}&\textsf{createObjAttr}(\text{Bob}, \texttt{oorg}, atomic, \texttt{vm})\\&\textsf{createObjAttr}(\text{Bob}, \texttt{oservice}, atomic, \texttt{vm})\\&\textsf{addObjConstr}(\text{Bob}, policy)\end{aligned}$$

where $policy$ is:

$$\begin{aligned}&\text{ConstrObj}(s, o, \{(\texttt{oorg}, porg), (\texttt{oservice}, pservice)\}) \equiv\\&\exists(org, service) \in \texttt{sorg\_service}(s).porg = org\wedge\\&service = pservice\end{aligned}$$

$$\textsf{addAuthz}(\text{Bob}, policy)$$

where $policy$ is:

$$\begin{aligned}\text{Authz}_{\textsf{restart\_instance}}(s, o) \equiv \exists(org, service) \in \texttt{org\_service}(s).\\ org = \texttt{oorg}(o) \wedge service = \texttt{oservice}(o)\end{aligned}$$

$$\begin{aligned}&\textsf{createAdminRole}(\text{Bob}, ITManager)\\&\textsf{createAdminPolicy}(\text{Bob}, policy_1)\\&\textsf{createAdminPolicy}(\text{Bob}, policy_2)\\&\textsf{createAdminPolicy}(\text{Bob}, policy_3)\end{aligned}$$

where $policy_1$ is:

$$\begin{aligned}\textsf{can\_add}_{\texttt{org\_service}} = \{(&ITManager, ITArchitect \in \texttt{role}(u),\\&\{(cs, web), (cs, email), (cs, app)\})\}\end{aligned}$$

$policy_2$ is:

$$\textsf{can\_add}_{\texttt{role}} = \{(ITManager, True, \{ITArchitect\})\}$$

and $policy_3$ is:

$$\textsf{can\_adduser} = \{ITManager\}$$

$$\begin{aligned}&\textsf{addAminUserRole}(\text{Bob}, Frank, ITManager)\\&\textsf{createUser}(\text{Frank}, \text{Gary})\\&\textsf{add}(\text{Frank}, \text{Gary}, \texttt{role}, ITArchitect)\\&\textsf{add}(\text{Frank}, \text{Gary}, \texttt{org\_service}, (cs, web))\\&\textsf{add}(\text{Frank}, \text{Gary}, \texttt{org\_service}, (ece, web))\end{aligned}$$

The complete list of operations is given in table IV.

TABLE IV: *Complete List of Functional Specifications*

Operations	**Updates**
1. Operations for Cloud Root User	
1.1 createTenant(req:CRU, tenant:NAME)	T′ = T ∪ {tenant}
1.2 createRootUser(req:CRU, u:NAME, tenant:T)	TRU = ∅, TRU = {u}
1.3 removeTenant(req:CRU, tenant:NAME)	T′ = T \ {tenant}
2. Operations for Tenant Root User	
2.1 createUserAttr(req:TRU, ua:NAME, type: {set, atomic})	UA′ = UA ∪ {ua}, attType(ua) = type
2.2 createUserAttrScope(req:TRU, ua:UA, value:NAME)	SCOPE'_{ua} = SCOPE_{ua} ∪ {value}
2.3 removeUserAttrScope(req:TRU, ua:UA, value:SCOPE_{ua})	SCOPE'_{ua} = SCOPE_{ua} \ {value}
2.4 createSubAttr(req:TRU, sa:NAME, type: {set, atomic})	SA′ = SA ∪ {sa}, attType(sa) = type
2.5 createSubAttrScope(req:TRU, sa:SA, value:NAME)	SCOPE'_{sa} = SCOPE_{sa} ∪ {value}
2.6 removeSubAttrScope(req:TRU, sa:NAME, value:SCOPE_{sa})	SCOPE'_{sa} = SCOPE_{sa} \ {value}
2.7 addSubConstr (req:TRU, policy:POLICY)	SubConstr′ = SubConstr ∪ {policy}
2.8 removeSubConstr (req:TRU, policy:POLICY)	SubConstr′ = SubConstr \ {policy}
2.9 createObjAttr (req:TRU, oa:NAME, type:{set, atomic}, oat: OT)	OA′ = OA ∪ {oa}, attType(oa) = type, oaType(oa) = oat
2.10 createObjAttrScope(req:TRU, oa:OA, value:NAME)	SCOPE'_{oa} = SCOPE_{oa} ∪ {value}
2.11 removeObjAttrScope(req:TRU, oa:OA, value:NAME)	SCOPE'_{sa} = SCOPE_{sa} \ {value}
2.12 addObjConstr (req:TRU, policy:POLICY)	ObjConstr′ = ObjConstr ∪ {policy}
2.13 removeObjConstr (req:TRU, policy:POLICY)	ObjConstr′ = ObjConstr \ {policy}
2.14 addAuthz (req:TRU, policy:POLICY)	Authz′ = Authz ∪ {policy}
2.15 removeAuthz (req:TRU, policy:POLICY)	Authz′ = Authz \ {policy}
2.16 createAdminRole(req:TRU, role:NAME)	AR′ = AR ∪ {role}
2.17 createAdminPolicy(req:TRU, policy:POLICY)	AdminPolicy′ = AdminPolicy ∪ {policy}
2.18 removeAdminPolicy(req:TRU, policy:POLICY)	AdminPolicy′ = AdminPolicy \ {policy}
2.19 addAminUser(req:TRU, u:NAME)	TAU′ = TAU ∪ { u}
2.20 removeAminUser(req:TRU, u:TAU)	TAU′ = TAU \ {u}
2.21 addAminUserRole(req:TRU, u:TAU, r:AR)	UAR′ = UAR ∪ {(u, r)}
2.22 removeAminUserRole(req:TRU, u:TAU, r:AR)	UAR′ = UAR \ {(u, r)}
3. Operations for Tenant Administrative Users [16]	
3.1 addUser(req:TAU, user:NAME)	TReU′ = TReU ∪ {user}
3.2 removeUser(req:TAU, user:TReU)	TReU′ = TReU \ {user}
3.3 add(req:TAU, tuser:TReU, att:UA, value:SCOPE_{att})	att(tuser)′ = att(tuser) ∪ {value}
3.4 delete(req:TAU, tuser:TReU, att:UA, value:SCOPE_{att})	att(tuser)′ = att(tuser) \ {value}
3.5 assign(req:TAU, tuser:TReU, att:UA, value:SCOPE_{att})	att(tuser)′ = value
4. Operations for Tenant Regular Users [17]	
4.1 createSubject(req:TReU, sub:NAME, saset:SASET)	S′ = S ∪ {sub}, for each (sa, val) ∈ SASET, sa(sub) = val, SubCreator(s) = req
4.2 createObject(sub:S, obj:NAME, oaset:OASET_t, t:OT)	O′ = O ∪ {obj}, objType(obj) = t, for each (oa, val) ∈ OASET, oa(obj) = val
4.3 modifyObjAttr(sub:S, obj:NAME, oaset:OASET)	For each (oa, val) ∈ OASET, oa(obj) = val
4.4 Operation(sub:S, obj:O)	None